Chapter 8

Binary Search Trees

Trees

Owner
Jake

Manager
Brad
Waitress
Joyce
Waiter
Chris

Chef
Carol
Cook
Max
Helper
Len

Jake’s Pizza Shop
Trees

**Binary Tree**
A structure with a unique starting node (the root), in which each node is capable of having two child nodes and a unique path exists from the root to every other node

**Root**
The top node of a tree structure; a node with no parent

**Leaf Node**
A tree node that has no children
Trees

Level
Distance of a node from the root

Height
The maximum level

Why is this not a tree?
Trees

How many leaf nodes?

Descendants

Definition:
A child of a node in a tree, any of the children of the children, etc.
Ancestors

Definition:
A parent of a node in a tree, the parent of the parent, etc.

V
Q
E
K
L
T
A
S

S has how many ancestors?

How many different binary trees can be made from 2 nodes? 4 nodes? 6 nodes?
Facts of Binary Tree

- Max. number of nodes at $N^{th}$ level: $2^N$.
- Max. total number of nodes with $N$ levels: $2^N - 1$
- Max. number of levels with $N$ nodes: $N$
- Min. number of levels with $N$ nodes: $\log_2 N + 1$

Binary Search Trees

**Binary Search Trees**

A binary tree in which the key value in any node is greater than the key value in the left child and any of its children and less than the key value in the right child and any of its children.

A BST is a binary tree with the search property.
Binary Search Trees

Each node is the root of a subtree

Recursive Implementation

Node

Left(Node) Right(Node)

Info(Node)
Recursive Count

Let’s start by counting the number of nodes in a tree

Size?

Base cases(s)?

General case(s)?

Recursive Count

Count (Version 1)

\[
\text{if (Left(tree) is NULL) AND (Right(tree) is NULL)} \\
\quad \text{return 1} \\
\text{else} \\
\quad \text{return } \text{Count}(\text{Left(tree)}) + \text{Count}(\text{Right(tree)}) + 1
\]

Apply to these trees?
Recursive Count

**Count (Version 2)**
if (left(tree) is NULL) AND (Right(tree) is NULL)  
return 1  
else if (Left(tree) is NULL)  
    return Count(Right(tree)) + 1  
else if (Right(tree) is NULL)  
    return Count(Left(tree)) + 1  
else return Count(Left(tree)) + Count(Right(tree)) + 1

Apply to an empty tree?

Recursive Count

**Count (Version 3)**
if (tree is NULL)  
   // Version 2

**Count (Version 4)**
if (tree is NULL)  
   return 0  
else  
   return Count(Left(tree)) + Count(Right(tree)) + 1
Recursive Count

```cpp
int TreeType::GetLength() const
{
    return Count(root);
}

int TreeType::Count(TreeNode* tree) const
{
    if (tree == NULL)
        return 0
    else
        return Count(tree->left) +
        Count(tree->right) + 1;
}
```

Why do we need two functions?

Shape of BST

- Shape depends on the order of item insertion
- Insert the elements 'J' 'E' 'F' 'T' 'A' in that order
- The first value inserted is always put in the root

'J'
Shape of BST

• Thereafter, each value to be inserted
  – compares itself to the value in the root node
  – moves left if it is less or
  – moves right if it is greater

• *When does the process stop?*

![Diagram of BST](image)

Shape of BST

• Trace path to insert ‘F’

![Diagram of BST](image)
Shape of BST

• Trace path to insert ‘T’

Shape of BST

• Trace path to insert ‘A’
Shape of BST

- Now build tree by inserting ‘A’ ‘E’ ‘F’ ‘J’ ‘T’ in that order

And the moral is?

Recursive Insertion

Insert an item into a tree

*Where does each new node get inserted?*

```
Insert(tree, item)
if (tree is NULL)
    Get a new node
    Set right and left to NULL
    Set info to item
else if item is larger than tree.info
    Insert(tree->right, item)
else
    Insert(tree->left, item)
```
Recursive Insertion

(a) The initial call

(b) The first recursive call

(c) The second recursive call

(d) The base case

Insert item 12
Recursive Insertion

void TreeType::Insert(TreeNode* & tree, ItemType item)
{
    if (tree == NULL)
    { // Insertion place found.
        tree = new TreeNode;
        tree->right = NULL;
        tree->left = NULL;
        tree->info = item;
    }
    else if (item.ComparedTo(tree->info) == LESS)
        Insert(tree->left, item);
    else
        Insert(tree->right, item);
}
Recursive Search

Are ‘D’, ‘Q’, and ‘N’ in the tree?

Recursive Search

Retrieve(tree, item, found)

Size?

Base case(s)?

General case(s)?
Recursive Search

void TreeType::Retrieve(TreeNode* tree, ItemType& item, bool& found) const {
    if (tree == NULL)  found = false; // base case
    else if (item.ComparedTo(tree->info) == LESS)
        Retrieve(tree->left, item, found);
    else if (item.ComparedTo(tree->info) == LARGER)
        Retrieve(tree->right, item, found);
    else // base case
    {
        item = tree->info;
        found = true;
    }
}

Recursive Deletion

Delete ‘Z’

Before:

```
  J
 / 
9   a
 / 
L   R
```

After:

```
  J
 / 
L   R
```

Delete the node containing Z
Recursive Deletion

Delete ‘R’

Before:

After:

Delete the node containing R

Recursive Deletion

Delete ‘Q’

Before:

After:

Delete the node containing Q
Predecessor

- **Predecessor**: the key of the element immediately precedes (less than) the key of item
  - If the item node has two children, the largest element in the left subtree (right-most child)

```
        G
         \
         D
          |
         B
          |
         A
          |
         C
          |
         E
          |
         F
           |
          G
```

Successor

- **Successor**: the key of the element immediately follows (greater than) the key of item
  - If the item node has two children, the smallest element in the right subtree (left-most child)

```
        G
         \
         D
          |
         B
          |
         A
          |
         C
          |
         E
          |
         F
           |
          G
```
Recursive Deletion

Delete an existing item:

Can you summarize the three deletion cases?

1. Deleting a leaf node.
2. Deleting a node with only one child.
3. Deleting a node with two children.

Recursive Deletion

```cpp
DeleteItem(tree, item)
if (Left(tree) is NULL) AND (Right(tree) is NULL) // delete 'Z'
    Set tree to NULL
else if Left(tree) is NULL // (Right(tree)) is not NULL, delete 'R'
    Set tree to Right(tree)
else if Right(tree) is NULL //(Left(tree)) is not NULL
    Set tree to Left(tree)
Else // delete 'Q', maintain a binary search tree
    Find predecessor
    Set Info(tree) to Info(predecessor)
    Delete predecessor
```
Recursive Deletion

- void Delete( TreeNode* & tree, ItemType item)
  Deletes item from tree.
- void DeleteNode( TreeNode* & tree)
  Deletes the matching node
- void GetPredecessor( TreeNode* tree, ItemType& data)
  Find data’s predecessor – the largest item in data’s left subtree and save the info into data.

Recursive Deletion

```cpp
// first, find which node should be deleted.
void TreeType::Delete(TreeNode* & tree, 
    ItemType item) 
{
    if (item < tree->info) 
        Delete(tree->left, item);
    else if (item > tree->info) 
        Delete(tree->right, item);
    else
        DeleteNode(tree);  // Node found
}
```
Recursive Deletion

```cpp
void TreeType::DeleteNode(TreeNode*& tree) {
    ItemType data;
    TreeNode* tempPtr;
    tempPtr = tree;
    if (tree->left == NULL) {
        tree = tree->right;
        delete tempPtr;
    } else if (tree->right == NULL) {
        tree = tree->left;
        delete tempPtr;
    } else {
        GetPredecessor(tree->left, data);
        tree->info = data;
        Delete(tree->left, data);
        //recursive call, Safe?
    }
}
```

Recursive Deletion

```cpp
void TreeType::GetPredecessor(TreeNode* tree, ItemType& data) {
    //the largest item is located in its rightmost node.
    while (tree->right != NULL) {
        tree = tree->right;
        data = tree->info;
    }
}
```

• *Why is the code not recursive?*
Recursive Deletion

Traversals

- Tree Traversal: visiting all the nodes of a tree
  - Depth-First Traversal, Breadth-First Traversal
- Depth-First Traversal
  - Inorder Traversal
  - Preorder Traversal
  - Postorder Traversal
Inorder Traversal

- Inorder traversal visits the root in between visiting the left and right subtrees:
  - Inorder traversal only makes sense for binary trees.

- **Inorder**(tree)
  
  if tree is not NULL
  1. *Inorder*(Left(tree))
  2. *Visit Info*(tree)
  3. *Inorder*(Right(tree))

Preorder Traversal

- Preorder traversal visits the root first.

- **PreOrder**(tree)
  
  if tree is not NULL
  1. *Visit Info*(tree)
  2. *Preorder*(Left(tree))
  3. *Preorder*(Right(tree))
Postorder Traversal

- Postorder traversal visits the root last.
- \textit{PostOrder(tree)}

\begin{verbatim}
if tree is not NULL
  1. Postorder(Left(tree))
  2. Postorder(Right(tree))
  3. Visit Info(tree)
\end{verbatim}
Traversal Algorithm: Inorder traversal

What is the order of the output?

PrintTree operation

Size?

Base case(s)?

General case(s)?
Printing the Tree

```cpp
void TreeType::PrintTree(TreeNode* tree, std::ofstream& outFile)
{
    if (tree != NULL)
    {
        PrintTree(tree->left, outFile);
        outFile << tree->info;
        PrintTree(tree->right, outFile);
    }
}
```

Is that all there is?

A Question

- *Do we need a destructor?*
- *Which Traversal Algorithm?*
  - *Postorder Traversal*
  - *Why?*
Copy a Tree

• Copy constructor
• Assignment operator = overloaded
• Use which traversal algorithm?
  – Preorder Traversal

Copy a Tree

\[
\text{CopyTree}(\text{copy}, \text{originalTree})
\]

\[
\begin{align*}
&\text{if} \ (\text{originalTree is NULL}) \\
&\quad \text{Set copy to NULL} \\
&\text{else} \\
&\quad \text{Set copy to new node} \\
&\quad \text{Set Info(copy) to Info(originalTree)} \\
&\quad \text{CopyTree(Left(copy), Left(originalTree))} \\
&\quad \text{CopyTree(Right(copy), Right(originalTree))}
\end{align*}
\]

How must copy be passed?
How must originalTree be passed?
Copying a Tree

```cpp
void CopyTree(TreeNode*& copy,
    const TreeNode* originalTree)
{
    if (originalTree == NULL)
        copy = NULL;
    else
    {
        copy = new TreeNode;
        copy->info = originalTree->info;
        CopyTree(copy->left, originalTree->left);
        CopyTree(copy->right, originalTree->right);
    }
}
```

Iterator

- **Approach**
  
The client program passes a parameter to `ResetTree` and `GetNextItem` indicating which of these three traversals to use
  - `ResetTree` generates a queues of node contents in the indicated order
  - `GetNextItem` processes the node contents from the appropriate queue: inQue, preQue, postQue
Iterator

void TreeType::ResetTree(OrderType order)
// Calls function to create a queue of the tree
// elements in the desired order.
{
    switch (order)
    {
        case PRE_ORDER : preQue = new QueueType;
            PreOrder(root, preQue);
            break;
        case IN_ORDER  : inQue = new QueueType;
            InOrder(root, inQue);
            break;
        case POST_ORDER: postQue = new QueueType;
            PostOrder(root, postQue);
            break;
    }
}

void TreeType::GetNextItem(ItemType& item,
OrderType order, bool& finished)
{
    finished = false;
    switch (order)
    {
        case PRE_ORDER : preQue->Dequeue(item);
            if (preQue->IsEmpty())
            {
                finished = true; delete preQue;
                preQue = NULL;
            }
            break;
        case IN_ORDER   : inQue->Dequeue(item);
            if (inQue->IsEmpty())
            {
                finished = true; delete inQue;
                inQue = NULL;
            }
            break;
        case POST_ORDER: postQue->Dequeue(item);
            if (postQue->IsEmpty())
            {
                finished = true; delete postQue;
                postQue = NULL;
            }
            break;
    }
}
Iterative Search  
a Binary Search Tree

- Searching an item
  - If a node is found with the same key as item’s, get two pointers, one points to the node and one points to its parent node.
  - If no node is found, the first pointers points to NULL and the second pointers points to its logical parent node.
  - If root is the node matching, the first pointers points to the root and the second pointer points to NULL.

Iterative Versions

```plaintext
FindNode(tree, item, nodePtr, parentPtr)
Set nodePtr to tree
Set parentPtr to NULL
Set found to false

while more elements to search AND NOT found
  if item < Info(nodePtr) // search left subtree
    Set parentPtr to nodePtr
    Set nodePtr to Left(nodePtr)
  else if item > Info(nodePtr) // search right subtree
    Set parentPtr to nodePtr
    Set nodePtr to Right(nodePtr)
  else //match
    Set found to true
```

Spring 2015  Yanjun Li CS2200
void TreeType::FindNode(TreeNode * tree, ItemType item, 
TreeNode * & nodePtr, TreeNode * & parentPtr) 
{
    nodePtr = tree;
    parentPtr = NULL;
    bool found = false;
    while( nodePtr != NULL & found == false)
    {
        if (item.ComparedTo(nodePtr->info) == LESS)
        {
            parentPtr = nodePtr;
            nodePtr = nodePtr->left;
        }else if (item.ComparedTo(nodePtr->info) == LARGER)
        {
            parentPtr = nodePtr;
            nodePtr = nodePtr->right;
        }else
        {
            found = true;
        }
    }
}

Iterative Versions

- **InsertItem**
  - Create a node to contain the new item
  - Find the insertion place
  - Attach new node

- **Find the insertion place**
  FindNode(tree, item, nodePtr, parentPtr);
Iterative Versions

(a) Insert 13

(b) Insert 13

(c) Insert 13

(d) Insert 13

Spring 2015 Yanjun Li CS2200
Iterative Versions

```
AttachNewNode
if item < Info(parentPtr)
    Set Left(parentPtr) to newNode
else
    Set Right(parentPtr) to newNode
```
Iterative Version

```c
void TreeType::DeleteItem(ItemType item)
{
    TreeNode* nodePtr;
    TreeNode* parentPtr;
    FindNode(root, item, nodePtr, parentPtr);
    if (nodePtr == root)
        DeleteNode(root);
    else
        if (parentPtr->left == nodePtr)
            DeleteNode(parentPtr->left);
        else DeleteNode(parentPtr->right);
}
```

Why not directly delete nodePtr?

parentPtr and nodePtr are external; parentPtr->left is internal
Recursion vs. Iteration

Compare versions of Tree algorithms

- Is the depth of recursion relatively shallow?
- Is the recursive solution shorter or cleaner?
- Is the recursive version much less efficient?

Nonlinked Representation

What is the mapping into the index?
Nonlinked Representation

• For any node `tree.nodes[index]`
  its left child is in `tree.nodes[index*2 + 1]`
  right child is in `tree.nodes[index*2 + 2]`
  its parent is in `tree.nodes[(index - 1)/2]`

• Can you determine which nodes are leaf nodes?

Nonlinked Representation

**Full Binary Tree**
A binary tree in which all of the leaves are on the same level and every nonleaf node has two children
Nonlinked Representation

Complete Binary Tree
A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.
Nonlinked Representation

A technique for storing a non-complete tree

Binary Search Trees

The same set of keys may have different BSTs

- Average depth of a node is $O(\log_2 N)$
- Maximum depth of a node is $O(N)$
Time Complexity

• Time complexity of Searching:
  – $O(\text{height of the tree})$

• Time complexity of Inserting:
  – $O(\text{height of the tree})$

• Time complexity of Deleting:
  – $O(\text{height of the tree})$

Comparison

• Assume that we have a Complete tree.

<table>
<thead>
<tr>
<th></th>
<th>Binary Search Tree</th>
<th>Array-based Linear List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Destructor</td>
<td>$O(N)$</td>
<td>$O(1)$ (non-pointers)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>MakeEmpty</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>GetLength</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IsFull</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>RetrieveItem</td>
<td>$O(\log_2 N)$</td>
<td>$O(\log_2 N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>InsertItem</td>
<td>$O(\log_2 N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>$O(\log_2 N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Reference

• Reproduced by permission of Jones and Bartlett Publishers International.